Reformation: A Domain-Independent Algorithm for Theory Repair

Reformation: A Domain-Independent Algorithm for Theory Repair *

Alan Bundy and Boris Mitrovic

University of Edinburgh,

A.Bundy@ed.ac.uk, borismitrovic@gmail.com

February 1, 2016

Abstract

We describe *reformation*, a new algorithm for the automated repair of faulty logical theories. A fault is revealed by a reasoning failure: either the proof of a false conjecture or the failure to prove a true conjecture. Repair suggestions are systematically extracted via analysis of (un)successful unifications of formulae in (broken) proofs. These suggestions will either unblock a wanted but unsuccessful unification attempt or block an unwanted but successful unification. In contrast to traditional abduction and belief revision mechanisms, the repairs are to the *language* of the theory as well as to the axioms. The intention is that the language repairs suggested by reformation complement these axiom deleting and adding repairs, adding to the overall capability of theory repair and evolution. Reformation is self-inverse in that any blocking repair can be undone by an unblocking one, and vice versa. This self-inverse property provides some assurance that reformation repairs are minimal.

1 Introduction

We are interested in how reasoning failures can trigger representational change. Moreover, we focus on *conceptual* changes, i.e., not just the addition or deletion of axioms in a theory, but changes in the *language* in which the axioms are expressed. Typical language changes include: the splitting of a function (or predicate) into two functions (or predicates); the merging of two distinct functions (or predicates) into one; and the change of the arity of a function (or predicate), e.g., by removing or adding one or more arguments.

The reasoning failures we consider in this paper take two forms: that a false conjecture has been proved; or that a true conjecture cannot be proved.

The key idea is to analyse the use of the unification algorithm during these proofs (or failed proof attempts) to suggest possible repairs to the language of the theory. We have developed a non-standard version of the first-order unification algorithm in which some of its steps are paired: each step leading to failure is paired with one leading to success. If one of these steps is triggered during a unification application, reformation can suggest one or more signature repairs that will cause the dual step to be triggered instead. The hypothesis[1] is that:

*Michael Chan, Szymon Klarman and two anonymous ECAI-12 referees gave useful feedback on an earlier version of this paper. The Edinburgh Mathematical Reasoning Group also gave helpful feedback during a seminar based on the paper. This work was supported by ONR grant N000140910467, ONR Global project N62909-12-1-7012, EPSRC project EP/J020524/1 and a Zois Scholarship from the Slovene Human Resources Development and Scholarship Fund to the second author.

[1]Given its simplicity and symmetry, I also think that it is beautiful, but that is in the eye of the beholder.

Reformation systematically generates reversible theory repairs that invert the outcome of unification.

This hypothesis is confirmed by theorems (2) and (3).

In this paper we describe first-order unsorted logic, but we have also adapted reformation to other logics, in particular: single and many-sorted first order logic [Mitrovic, 2013] and the $\mathcal{ALC}$ description logic [Tsialos, 2015].

2 Motivating Examples

Below we give some examples that motivate the need for language changes driven by reasoning failures.

2.1 The KnowItAll Theory

The KnowItAll Theory has been automatically formed by parsing natural language text on the web into about 6 billion RDF triples plus about 30 thousand rules [Etzioni *et al*, 2011]. The authors claim 90% accuracy[2], but this means about 600 million errors, which it would be impossible to correct manually. In [Gkaniatsou *et al*, 2012] we conducted a preliminary exploration of whether these errors could be automatically detected and, hence, repaired. A very small subset of the ontology was translated into the TPTP format [Sutcliffe *et al*, 1994] and the E prover [Schulz, 2002] applied to it to detect inconsistencies. These inconsistencies were analysed and classified as a preliminary to automatic repair of the errors.

One difficulty was that the KnowItAll Theory does not contain negation, so E could not be straightforwardly used to prove the empty clause. As an alternative, E was used to detect violations of rules that asserted a uniqueness property of the i^{th} argument of a predicate P, i.e., that for each combination of values of the other arguments there is at most one value for the i^{th} argument. An example such rule is:

$$Cap\_of(x, z) \wedge Cap\_of(y, z) \quad \Longrightarrow \quad x = y$$

where $Cap\_of(x, z)$[3] means that city x is the capital of country z.

As examples of the kinds of violations found, the ontology contains: the true $Cap\_of(Tokyo, Japan)$, but also the false $Cap\_of(Kyoto, Japan)$ and $Cap\_of(Paris, Japan)$. One can track where these two false triples came from. The $Paris$ one came from a tutorial on logic and arose from an example of a false assertion. The repair here would be just to delete the false triple, as in belief revision §10. The $Kyoto$ one, however, came from an assertion that "Kyoto *was* the capital of Japan", which is true. It was normalised into the present tense as part of the KnowItAll formalisation process. Rather than delete such false triples, it would be preferably to repair them into true ones.

For instance, either:

- a past tense variant of the Cap_of predicate might be created and used to replace $Cap\_of(Kyoto, Japan)$ with, say, $was\_Cap\_of(Kyoto, Japan)$; or
- an extra time argument might be added to the Cap_of predicate so that the two retained triples become, say, $Cap\_of(Tokyo, Japan, Present)$ and $Cap\_of(Kyoto, Japan, Past)$.

2.2 ORS

The Ontology[4] Repair System (ORS) uses patterns of failures during the execution of a plan to diagnose and repair the theory used to form that plan [McNeill & Bundy, 2007]. The services

[2]But this refers to accuracy of the *translation*, not of the text from which the triples are extracted.

[3]The actual KnowItAll predicate was $be\_the\_capital\_of$ rather than Cap_of, but we've shortened it to aid presentation.

[4]We will use "ontology" as a synonym for logical theory.

provided by online agents are represented by STRIPS-like planning operators, consisting of preconditions and effects. The planning agent (PA) forms plans to combine the services of multiple, service-providing agents (SPAs) based on its representations of the preconditions under which they will provide services and the effects of their doing so. The PA's representations of an SPA operators may, however, differ from those of the SPA's own representations. In this case, the plan formed may fail when executed. Since the SPA can be regarded as the indisputable authority on the services it provides, it is the PA's representation that must be repaired to restore consistency. Multiple repairs may be required before a successful plan is formed.

The ORS diagnosis process works by considering a failed execution as a failure to prove a true conjecture. It then tries to repair the PA's ontology to unblock this failed proof attempt. One of the diagnostic heuristics it uses is the receipt of *surprising questions*. For instance, the PA might expect the SPA to ask it whether it is willing to pay for the service it has requested. It might expect this query to take the form of the goal $Pay(PA, £200, SPA)$. Suppose, though, that it actually receives the goal $Pay(PA, £200, SPA, Credit\_Card)$. The PA must now modify its representation, namely by adding a fourth argument to Pay and by instantiating it to $Credit\_Card$.

2.3 Repairing a Faulty Proof of Cauchy's

Theory repair can also be applied to mathematics. In the following example a failed proof attempt suggests both defining a new concept and adapting the conjecture to be proved. Our example is drawn from [Lakatos, 1976][Appendix I]. The main text of Lakatos's book gives a rational reconstruction of the history of Euler's Theorem about the relationship between the sides, edges and vertexes of polyhedra, but this appendix describes a failed proof, due to Cauchy, that the limit of a convergent series of continuous functions is itself continuous. For some time, a counter-example to this faulty theorem was known. Fourier analysis provides a convergent series of (continuous) sine functions whose limit is the (discontinuous) square wave. Despite this, mathematicians of the time had difficulty identifying the fault in Cauchy's 'proof'.

The repair, when it was finally discovered, was to replace 'convergence' with '*uniform* convergence' in the theorem's premise. These two concepts differ only in the order of their quantifiers.

$$\text{Convergence:} \qquad \forall x.\forall \epsilon > 0.\exists m.\forall n \geq m.\ |\textstyle\sum_i f_i(x)| < \epsilon$$

$$\text{Uniform Convergence:} \quad \forall \epsilon > 0.\exists m.\forall x.\forall n \geq m.\ |\textstyle\sum_i f_i(x)| < \epsilon$$

Note that the second definition can be generated from the first just by moving the $\forall x$ from being the first quantifier in convergence to being the third in uniform convergence.

[Bundy, 1985] describes a mechanism for automatically analysing the failed proof and repairing it by generating the definition of uniform convergence and correcting the false conjecture to a true one, so that the failed proof attempt then succeeds. In modern terms, the error in Cauchy's proof can, be described as an *occurs check* omission during the unification of two expressions. During the failed proof there is an attempt to unify:

$$y \geq y \quad \text{with} \quad n \geq M(x + B(\Delta(E/3, x, n)), E/3) \tag{1}$$

where M, E, B and Δ are skolem functions arising from the existentially quantified variables m, ϵ, b and δ in either the definition of convergence or of continuity.

$$\text{Continuity:} \quad \forall x.\forall \epsilon > 0.\exists \delta > 0.\forall b.\ |b| < \delta \implies |f(x + b) - f(x)| < \epsilon$$

The standard unification algorithm is described in Table 1. Note that, when this algorithm is applied to unification problem (1), the $Occurs$ condition $n \in \mathcal{V}(M(x + B(\Delta(E/3, x, n)), E/3))$ is true, i.e., n occurs as a free variable in $M(x + B(\Delta(E/3, x, n)), E/3)$.

This faulty proof can be repaired in at least three different ways, each of which unblocks this unsuccessful unification. The embedded n must be removed from $M(x + B(\Delta(E/3, x, n)), E/3)$.

This unwanted n is embedded in three skolem functions: M, B and Δ: it is embedded in the first argument of M, the only argument of B and the third argument of Δ. If any one of these three arguments were removed, then n would no longer occur in $M(x+B(\Delta(E/3,x,n)),E/3)$, the *Occurs* condition would be false and the failed proof would succeed. Skolem functions are created from existentially quantified variables and inherit their arguments from the universal quantified variables that precede that existential quantifier. Thus the offending arguments can be removed by reordering these quantifiers in either the definition of convergence or of continuity.

- M comes from the existential variable m in the definition of convergence. To remove its first argument, the $\forall x$ must be moved after $\exists m$. This creates the definition of uniform convergence. Now the previously faulty resolution step is replaced by:

  $$y \geq y \quad \text{with} \quad n \geq M(E/3)$$

  which succeeds since $n \notin \mathcal{V}(M(E/3))$.

- B is a skolem function that arises from the *universal* variable b in the definition of continuity. This particular definition of continuity comes from the conjecture, so is negated. Thus turns the universal variable b into an existential one, and the existential variable δ into a universal one. To remove B's argument, the $\exists\delta$ must be moved after $\forall b$. This creates the definition:

  $$\forall x.\forall \epsilon > 0.\forall b.\exists \delta > 0.\ |b| < \delta \implies |f(x+b) - f(x)| < \epsilon$$

  This repair also succeeds, but the adapted definition of continuity holds for all functions, so the resulting theorem is uninteresting.

- Δ is a skolem function that arises from the existential variable δ in the definition of continuity in the following form:

  $$\forall n \in \mathbb{N}.\forall x.\forall \epsilon > 0.\exists \delta.\forall b.\ |b| < \delta \implies |S(n,x+b) - S(n,x)| < \epsilon$$

  where $S(n,x)$ is the finite sum of the f_i up to n, i.e., $\sum_i^n f_i(x)$. Moving $\forall n \in \mathbb{N}$ after $\exists\delta$ gives the following variant of continuity:

  $$\forall x.\forall \epsilon > 0.\exists \delta.\forall n \in \mathbb{N}.\forall b.\ |b| < \delta \implies |S(n,x+b) - S(n,x)| < \epsilon$$

  This concept was independently discovered by Ascoli, who called it *equi-continuity* [Ascoli, 1883]. It suggests a different and interesting repair of Cauchy's theorem in which the f_i are equi-continuous. Ascoli proved a different theorem about equi-continuity.

3 Deduction as Resolution in Clausal, First-Order Theories

In the rest of this paper we will assume that the axioms and theorems of a theory consist of FOL clauses and that inference is conducted by resolution [Robinson, 1965]. Our methodology is, however, applicable to any logical theory in which inference employs a unification algorithm.

3.1 Clausal Form

Clausal form is a normal form of first-order formulae in which they are rewritten into a weakly equivalent[5] conjunction of clauses.

Definition 1 (Clause) *A* clause *is a disjunction of literals. The empty clause will be represented by* $\square$*, which is interpreted as false. A* literal *is a negated or unnegated proposition. A* proposition *consists of an n-ary predicate applied to n terms, where n is a natural number. A* term *consists of either a variable or an n-ary function applied to n terms. When n is zero, the function or*

[5] *Weakly equivalent* means that the clauses have a model iff the original formula does.

predicate is called a constant*. For convenience, we will use the word* functor *to include both functions and predicates and* expression *to include all kinds of logical expression. We will use e, possibly subscripted, to represent expressions. The* axioms *are a set of clauses defining a logical theory and assumed to be true.*

We will adopt the following convenient notational conventions.

- $[n]$ will represent the set of natural numbers $\{1, \ldots, n\}$.
- Functors start with upper case letters and variables with lower case ones.
- We will use vectors to represent the arguments of n-ary functors, i.e., $F(t_1, \ldots, t_n)$ will be represented by $F(\vec{t}^n)$. Note the non-standard use of the superscript n to indicate the length of the vector.
- It is standard to represent conjunctions of clauses as sets, which we will do, but we will represent each clause as a disjunction of literals.
- Wlog, we will adopt the convention that the last literal in a disjunction is the one to be resolved on next and write $C \vee L$, where L is the literal being resolved and C is the disjunction of the remaining literals.

3.2 Unification

Definition 2 (Substitution) *A* substitution *is a set of pairs x_i/s_i, for $i \in [n]$, where the x_i are distinct variables and the s_i are terms that contain none of the x_j for $j \in [n]$. This restriction avoids non-termination and ensures that the substitution only needs to be applied once.*

We will use σ, possibly subscripted, to represent substitutions. We will write $e\sigma$ to represent the application of substitution σ to expression e, i.e., the replacement of each x_i in e with s_i. We will write $\sigma_1 \oplus \sigma_2$ to represent the composition of substitutions σ_1 and σ_2, such that $e(\sigma_1 \oplus \sigma_2)$ is identical to $(e\sigma_1)\sigma_2$ for all expressions e.

Definition 3 (Free Variables) *The* free variables $\mathcal{V}(t)$ *of a term t is a set of variables, which is defined recursively as follows:*

$$\begin{aligned} \mathcal{V}(x) &:= \{x\} \qquad \text{where } x \text{ is a variable} \\ \mathcal{V}(F(\vec{t}^n)) &:= \bigcup_{i=1}^{n} \mathcal{V}(t_i) \\ \mathcal{V}(\mathcal{Q}x.P) &:= \mathcal{V}(P) \setminus \{x\} \text{where } \mathcal{Q} \text{ is } \forall \text{ or } \exists \end{aligned}$$

Note that $\mathcal{V}(C) = \{\}$ when C is a constant, i.e., a nullary functor.

Definition 4 (Unification Problem) *A* unification problem *is a conjunction $e_1 \equiv e'_1 \wedge \ldots \wedge e_n \equiv e'_n$, where the e_i and e'_i are expressions. A* solution *to a unification problem is a* most general unifier *(mgu). A* unifier *of such a problem is a substitution σ such that $e_i\sigma$ is identical to $e'_i\sigma$ for all $i \in [n]$. A* mgu *is a unifier σ such that for any other unifier σ' there exists a substitution σ'' such that $\sigma' = \sigma \oplus \sigma''$. Mgus in* FOL *are unique up to renaming of variables.*

We will use u, possibly subscripted, to range over unification problems.

Table 1 describes the standard unsorted first-order unification algorithm, which has been adapted from [Baader & Snyder, 2001][p455]. As is now standard, the presentation is a collection of conditional transformation rules. Given a unification problem matching the **Before** pattern for which the **Condition** is true, the algorithm transforms the problem into the **After** pattern and then recurses. Given a unification problem, this algorithm returns its unique mgu. To reduce clutter the well-formedness conditions are expressed implicitly via the naming conventions described in the Table's caption.

Case	**Before**	**Condition**	**After**
Base	$\top; \sigma$		Terminates
Trivial	$s \equiv s \wedge u; \sigma$		$u; \sigma$
Decomp	$F(\vec{s^n}) \equiv F(\vec{t^n}) \wedge u; \sigma$		$\bigwedge_{i=1}^{n} s_i \equiv t_i \wedge u; \sigma$
Clash	$F(\vec{s^m}) \equiv G(\vec{t^n}) \wedge u; \sigma$	$F \neq G \vee m \neq n$	fail
Orient	$t \equiv x \wedge u; \sigma$		$x \equiv t \wedge u; \sigma$
Occurs	$x \equiv s \wedge u; \sigma$	$x \in \mathcal{V}(s) \wedge x \neq s$	fail
Var Elim	$x \equiv s \wedge u; \sigma$	$x \notin \mathcal{V}(s)$	$u\{x/s\}; \sigma \oplus \{x/s\}$

Table 1: **Standard Unsorted Unification Algorithm**: *F and G are functors; t is a non-variable term and s is any term; s_i and t_j are terms; $m; n \geq 0$; x and y are distinct variables; and u is a unification problem, $s \equiv t$ is the problem of unifying s and t, and $\mathcal{V}(t)$ is the set of free variables in t. The algorithm terminates with success and returns σ when the Before state is $\top; \sigma$, where $\top$ is the empty conjunction. The Trivial step makes the algorithm non-deterministic, since it will apply when Decomp is also applicable. Trivial is usually applied eagerly for efficiency reasons.*

3.3 Resolution

Resolution proofs are usually by refutation. The conjecture to be proved is negated and put into clausal form. These conjecture's clauses are added to the clauses obtained by converting the axioms of the logical theory into clausal form. If $\square$ can be proved from this conjunction, then the conjecture is a theorem. We will call the union of the conjecture clauses and the axiom clauses the *initial clauses*[6].

Definition 5 ((Broken) Derivation) *We define a* resolution derivation *by recursion. We will use* binary resolution*, i.e., two literals are resolved at each resolution step*[7]*. For completeness, we also need a* binary factoring *rule, in which a pair of literals in a clause are merged. So, there is one base case (initial clauses) and two step cases (resolution and factoring) in the recursive definition.*

Initial Clauses: *If C is an initial clause then $Initial(C)$ is a derivation with* conclusion *C.*

Binary Resolution: *Let π_1 and π_2 be derivations with conclusions $C_1 \vee P_1$ and $C_2 \vee \neg P_2$, respectively, where the P_i are propositions. If the mgu of P_1 and P_2 is σ, then $Resolve(\pi_1, \pi_2, (C_1 \vee C_2)\sigma)$ is a derivation with conclusion $(C_1 \vee C_2)\sigma$.*

Binary Factoring: *Let π be a derivation with conclusion $C \vee L_1 \vee L_2$, where the L_i are literals. If the mgu of L_1 and L_2 is σ, then $Factor(\pi, (C \vee L_1)\sigma)$ is a derivation with conclusion $(C \vee L_1)\sigma$.*

A derivation is a proof *iff its conclusion is $\square$.*

[6] Even when there are no conjecture clauses, i.e., when we are showing that the axioms are inconsistent.

[7] Alternatively, we could use n-ary resolution, where 2 or more literals are simultaneously unified. Then we don't need the factoring rule. Since reformation is defined for pairs of expressions, however, it is simpler to use these binary rules.

A broken derivation (proof) *has the same form as a derivation (proof) except that one or more of the substitutions σ is not a mgu of the literals it is supposed to unify.*

Binary resolution can be illustrated using an example from §2.1.

$$\frac{Cap\_of(Kyoto, Japan), \quad \neg Cap\_of(y, z) \vee x = y \vee \neg Cap\_of(x, z)}{\neg Cap\_of(y, Japan) \vee Kyoto = y} \tag{2}$$

where the literals being resolved are highlighted in mahogany and the literals being carried forward to the resolvant are highlighted in olive green.

Reformation will systematically generate repairs that (a) block successful but unwanted resolution applications and (b) unblock unsuccessful but wanted resolution applications. It achieves this by causing unification to either (a) fail or (b) succeed, where it would otherwise have (a) succeeded or (b) failed.

4 Non-Standard Unification Algorithms

Reformation is adapted from unification. As a prelude to defining reformation, we first define a non-standard unification algorithm which acts as an intermediate staging point.

Table 2 describes a slightly non-standard version of the unsorted FOL unification algorithm. The transformation rules are collected into three cases: compound terms *vs* compound term (CC), variable *vs* compound terms (VC) and variable *vs* variable (VV). The subscripts s and F indicate whether the rule results in success or failure.

Case	**Before**	**Condition**	**After**
Base	$\top; \sigma$		Terminates
CC_s	$F(\vec{s}^m) \equiv G(\vec{t}^n)$	$F = G \wedge n = m$	$\bigwedge_{i=1}^{n} s_i \equiv t_i \wedge u; \sigma$
CC_f	$\wedge u; \sigma$	$F \neq G \vee n \neq m$	Fail
VC_f	$x \equiv t \wedge u; \sigma$	$x \in \mathcal{V}(t)$	Fail
VC_s	or $t \equiv x \wedge u; \sigma$	$x \notin \mathcal{V}(t)$	$u\{x/t\}; \sigma \oplus \{x/t\}$
$VV_=$	$x \equiv x \wedge u; \sigma$		$u; \sigma$
$VV_\neq$	$x \equiv y \wedge u; \sigma$	$x \neq y$	$u\{x/y\}; \sigma \oplus \{x/y\}$

Table 2: **Our Non-Standard Unification Algorithm.** *The notational conventions are as for Table 1. CC names a rule that is applicable when the inputs are both compound terms or constants, and VC when just one of the inputs is a variable. Their s and f subscripts indicate rules resulting in success and failure, respectively. Note that there is no counterpart of the Trivial step because it would skip over Decomp steps that reformation might want to block. If we ignore the non-determinism of the order in which conjuncts are tackled, this unification algorithm is deterministic: exactly one step applies to each conjunct.*

4.1 Unification Algorithm Equivalence

Table 2 presented a non-standard version of the unification algorithm, which we will call ns_unify. To suit the purposes of the reformation algorithm, it was adapted from the standard unification

algorithm in Table 1, which we will call s_unify. We need to show that these two algorithms are equivalent, i.e., that ns_unify will unify two expressions iff s_unify will, and that they will return the equivalent unifiers in the event of success.

The theorem we have to prove is:

Theorem 1 (Equivalence of Unification Algorithms)

$$ns\_unify(u;\sigma) \quad = \quad s\_unify(u;\sigma)$$

Proof Sketch: *Note that both unification algorithms terminate. The following well-founded order is decreased by each of the unification steps. It is a lexicographical combination of the following three measures of the conjunction of unification problems: the number of variables with multiple occurrences, the number of functors on the left-hand sides of equivalences and the number of conjuncts. This also means that induction on the recursive structures of the algorithms is valid.*

The proof that these two algorithms return equivalent results is by induction on their recursive structures. The base case is $\top;\sigma$, which is the termination condition for both algorithms and they return the same output. In the step case the Before formula is $e_1 \equiv e_2 \wedge u;\sigma$. The idea of the proof is to show that for each step of one unification algorithm there are one or more corresponding steps in the other algorithm. These correspondences can be summarised in the following two tables:

ns_unify	s_unify
CC_s	$Decomp$
CC_f	$Clash$
VC_f	$Occurs + Orient$
VC_s	$Var\ Elim + Orient$
$VV_=$	$Trivial$
$VV_\neq$	$Var\ Elim$

s_unify	ns_unify
$Trivial$	$VV_= + CC_s$
$Decomp$	CC_s
$Clash$	CC_f
$Orient$	$VC_f + VC_s$
$Occurs$	$VC_f + VV_=$
$Var\ Elim$	$VV_\neq + VC_s$

The left-hand table gives the s_unify steps corresponding to each ns_unify step and the right-hand table gives the opposite direction. The + indicates that more than one step is required: sometimes to deal with different cases, sometimes in combination, sometimes repeatedly. QED

5 Reformation

Note that steps of the unification algorithm in Table 2 are paired. Each pair shares the same **Before** pattern but they have complementary **Condition**s — one resulting in success and one resulting in failure. By inverting these conditions a successful step can be turned into a failure and vice versa. For instance:

- An application of the successful step CC_s in Table 2 can be blocked either by splitting F and G into different functors or by adding extra but different arguments to each of them. On the other hand, an application of the unsuccessful step CC_f can be unblocked by ensuring that F and G are the same functor with the same arity.
- An application of the successful step VC_s in Table 2 can be blocked by including x as a subterm of $F(\vec{s}^m)$. On the other hand, an application of the unsuccessful step VC_f can be unblocked by removing x as a subterm of $F(\vec{s}^m)$.

The reformation algorithm is a modified version of the non-standard unification algorithms described in Tables 2, but with the following changes:

- Reformation carries an additional argument: whether we *want* the unification to succeed or fail.
- If the result of the unification *meets* this requirement, then return either a unifying substitution or 'fail', as appropriate.
- If the result of the unification *differs from* this requirement, then some repair suggestions are produced as to how to modify one or both of the two expressions to be unified in order to meet the required result.
- Each blocking repair is itself sufficient to cause unification to fail, so blocking returns a disjunction of atomic repairs. On the other hand, as illustrated in §9.3, multiple atomic repairs may be needed to completely unblock a failed unification, so unblocking returns a conjunction of atomic repairs.
- There are sometimes multiple ways to realise a repair. The details are explained in §8.3. The different realisations are returned as a disjunction. In the case of blocking these choices merely increase the length of the disjunction. In the case of unblocking, however, we now have a conjunction of disjunctions, whereas what we want is a disjunction of conjunctions. Each conjunction defines a different compound repair and the disjunction of them provides a choice of which compound repair to apply. In §8.3, we describe how our Prolog implementation uses backtracking efficiently to convert a conjunction of disjunctions of atomic repairs into a disjunction of conjunctions.
- In order to generate *all* repair suggestions, the unification must be continued until it succeeds. In the case of blocking, this means that none of the repair suggestions should be applied until the unification has successfully concluded. In the case of unblocking, on the other hand, this means that each of the repair suggestions should be immediately applied, so that the unification attempt no longer fails but, in case there are multiple reasons for failure, is allowed to continue to eventual success.

Table 3 page 10 describes the reformation algorithm.

5.1 Realising Atomic Repairs

Note that the atomic repair operations, i.e., those that are not recursive calls to $Block/Unblock$, are highlighted in red or blue. The two red repairs are duals of each other, i.e., each inverts the repair of the other. Ditto the two blue repairs. This observation is at the heart of the proof of theorem (3) in §6.1) .

Note that the recursive calls to $Block$ and $Unblock$ are there just to provide the non-deterministic choice to apply the repair anywhere within the expressions being unified. One cannot choose these recursive calls indefinitely. Eventually, they will be null choices and an atomic repair must be chosen.

Each atomic repair can be realised in multiple ways[8]. These realisations are applied locally to the (broken) proofs and not to the theories. In §8 we describe how these local repairs can be propagated to the whole theory and/or the conjecture. Below we give completely unrestricted ways of realising these atomic repairs. These unrestricted versions are needed for the proof of theorem (3), but define an infinite search space. In §7.1 we describe heuristics for restricting these realisations in order to make reformation computationally feasible.

The realisations are outlined as follows:

[8]So, our atomic repairs are not really atomic, but then neither are atoms in Physics.

Case	**Before**	**Condition**	**Block**	**Unblock**
Base	$\top$		Failure	Success
CC_s	$F(\vec{s}^m) \equiv G(\vec{t}^n)$ $\wedge u$	$F = G$ $\wedge\ m = n$	Make $F(\vec{s}^m) \neq F(\vec{t}^m)$ $\bigvee_{i=1}^{n}$ Block $s_i \equiv t_i$ $\vee$ Block u	$\bigwedge_{i=1}^{n}$ Unblock $s_i \equiv t_i$ $\wedge$ Unblock u
CC_f		$F \neq G$ $\vee\ m \neq n$	Success	Make $F(\vec{s}^m) = G(\vec{t}^n)$ $\bigwedge_{i=1}^{n}$ Unblock $\nu(s_i) \equiv \nu(t_i)$ $\wedge$ Unblock $\nu(u)$
VC_s	$x \equiv t \wedge u$ or $t \equiv x \wedge u$	$x \notin \mathcal{V}(t)$	Make $x \in \mathcal{V}(t)$ $\vee$ Block $u\{x/t\}$	Unblock $u\{x/t\}$
VC_f		$x \in \mathcal{V}(t)$	Success	Make $x \notin \mathcal{V}(t)$ $\wedge$ Unblock $\nu(u\{x/t\})$

Table 3: **The Reformation Algorithm for First-Order Logic.** *The notational conventions are as for Tables 1 and 2. The* **Block** *column gives a disjunction of repair suggestions any of which will block unification. The* **Unblock** *column gives a disjunction of conjunctions of repair suggestions; The whole of each conjunction is required to unblock unification. Any repair at an Unblock step must be applied to any expressions being recursively unblocked. We use the notation $\nu(e)$ (Pronounced "new e") to represent the application of the repair ν to expression e. Atomic repairs are highlighted in red or blue. The Block and Unblock commands are recursive calls to reformation with the appropriate parameter setting. Substitutions are omitted to reduce clutter.*

Make $F(\vec{s}^m) \neq F(\vec{t}^m)$: We can split the functor F into two, say F_1 and F_2, and/or make the arities m different by adding and/or deleting arguments from either functor. If, in order to make the arities different, we choose to add arguments to one of the functors, then we can choose any well-formed term for any new argument.

Make $F(\vec{s}^m) = G(\vec{t}^n)$: We need to merge both the functor names and the arities, i.e., make $F \equiv G$ and $m \equiv n$. The functor merges are equivalent up to renaming, but the arity changed could be realised by adding and/or deleting arguments from either functor. If, in order to make the arities equal, we choose to add arguments to one of the functors, then we can choose any well-formed term for any new argument.

Make $x \in \mathcal{V}(t)$: We can add x as an additional argument to any of the functors contained in t. For instance, if $F(t_1, \ldots, t_n)$ is contained in t then we can add one or more copies of x as additional arguments.

Make $x \notin \mathcal{V}(t)$: We must remove all occurrences of x from t. For instance, if $F(t_1, \ldots, t_n)$ is contained in t and t_i contains x, we can reduce the arity of F by one by removing the i^{th} argument, then recurse to remove the remaining occurrences of x.

5.2 Soundness of Reformation

Theorem (2) below supplies part of the successful theoretical evaluation of the hypothesis in §1 The other part is provided by theorem (3) in §6.1. Supplementary empirical evidence is provided by the implementation outlined in §7.

Theorem 2 (Soundness of Reformation)

$$\forall u : \mathbb{U}.\ \neg Trivial(u) \wedge Unify(u) \neq Want(u) \rightarrow$$
$$\forall \nu : \mathbb{V}.\ \nu \in Reform(u, Want(u)) \quad \rightarrow \quad Unify(\nu(u)) = Want(u)$$

where: $\mathbb{U}$ *is the type of unification problems;* $\mathbb{V}$ *is the type of repair suggestions, i.e., mappings that repair unification problems;* $Trivial(u)$ *means that* u *is the empty conjunct of unification problem or consists only of identities between two variables;* $Want(u)$ *is the desired result of the unification problem* u*;* $Unify(u)$ *is the actual result of the unification problem* u*; and* $Reform(u, w)$ *is the set of suggested repairs.*

Proof Sketch: *The proof has two main cases:* **Block** *and* **Unblock**. *Each of these cases is proven by induction on the recursive structure of the algorithm. The base cases of each of these inductions is degenerately true, as an empty conjunct is trivial. We now summarise the two step cases.*

Block: *Since the conjunction of unification problems is non-trivial and the unification is initially successful, then there exists a conjunct to which either step* CC_s *or step* VC_s *applies. Reformation will suggest a repair to this step that would cause the unification to fail. The recursive application of reformation may suggest further such repairs.*

Unblock: *Since the conjunction of unification problems is non-trivial, then there exists a conjunct to which either a step of type* CC *or* VC *applies. Since the unification initially fails, then either one of these is a failure step (*CC_f *or* VC_f*) or a failure happens during a recursive call of reformation, in which case we can invoke the induction hypothesis to provide the necessary unblocking repairs. These repairs must be conjoined with any repairs arising from steps* CC_f *or* VC_f *in the current call.*

6 Reformation is Self-inverse

Any reformation repair can be undone by applying reformation again. In particular, for every blocking repair there is an inverse unblocking one, and vice versa. This shows that no information is lost during an application of reformation, which shows that reformation repairs are in some sense *minimal*.

6.1 The Self Inverse Proof

Theorem 3 (*Block* and *Unblock* are Inverses)

$$Block(\pi_1, \pi_2) \quad \Longleftrightarrow \quad Unblock(\pi_2, \pi_1)$$

where π_1 *is a possibly broken proof and* π_2 *is a broken proof.*

- $Block(\pi_1, \pi_2)$ *means that broken proof* π_2 *is a possible repair of proof* π_1 *after applying reformation to block it.*
- $Unblock(\pi_2, \pi_1)$ *means that (broken) proof* π_1 *is a possible repair*[9] *of broken proof* π_2 *after applying reformation to unblock it.*

Note that Block and Unblock are binary predicates rather than unary functions, because the reformation repair of a faulty proof is not uniquely determined.

[9] Note that π_1 may still be broken since π_2 may contain several blockages.

Proof *The proof is by two cases:* $Block(\pi_1, \pi_2) \implies Unblock(\pi_2, \pi_1)$ *and* $Unblock(\pi_2, \pi_1) \implies Block(\pi_1, \pi_2)$.

$\underline{Block(\pi_1, \pi_2) \implies Unblock(\pi_2, \pi_1)}$ *We assume that an unwanted proof* π_1*, e.g., of the inconsistency of the axioms, has been blocked by reformation, resulting in a broken proof* π_2.

To block π_1*, one of the following blocking atomic repairs must have been applied to a resolution or factoring step.*

- *Make* $F(\vec{s}^m) \neq F(\vec{t}^m)$ *to the successful unification problem* $F(\vec{s}^m) \equiv F(\vec{t}^m)$.
- *Make* $x \in \mathcal{V}(t)$ *to the successful unification problem* $x \equiv t$ *(or* $t \equiv x$*).*

Suppose, wlog, that Block had blocked the unification of L_1 *and* L_2*, so that* σ *no longer unifies them. Suppose this broken proof* π_2 *is submitted to Unblock. It is able to use either of the following atomic repairs to unblock this failed unification.*

- *Make* $F(\vec{s}^m) = G(\vec{t}^n)$ *to the failed unification problem* $F(\vec{s}^m) \equiv G(\vec{t}^n)$.
- *Make* $x \notin \mathcal{V}(t)$ *to the failed unification problem* $x \equiv t$ *(or* $t \equiv x$*).*

Note that each of these unblocking atomic repairs is the dual of one of the blocking atomic repairs. So, one of these two repairs can exactly undo the blocking repair and restore the unification problem to its previous successful state. Note that there are multiple realisations of these unblocking steps, but we only need to consider the one that exactly reverses the previous blocking repair and ignore the others. We know what that reversing repair is as we have the target proof as well as the source one.

We consider the two cases in more detail:

Make $F(\vec{s}^m) \neq F(\vec{t}^m)$**:** π_1 *has been blocked by renaming* F *to* G *in* L_2*, say, and by changing the arities of* F*,* G *or both. The unblocking repair Make* $F(\vec{s}^m) = G(\vec{t}^m)$ *can rename* G *back to* F *and remove any arguments that were added and put back any that were removed.*

Make $x \in \mathcal{V}(t)$**:** π_1 *has been blocked by adding* x *zero or more times to each* F *in* t*. The unblocking repair Make* $x \notin \mathcal{V}(t)$ *will remove each of these occurrences of* x *from* t.

$\underline{Unblock(\pi_2, \pi_1) \implies Block(\pi_1, \pi_2)}$ *We assume that a failed proof* π_2*, e.g., of a true theorem, has been unblocked by reformation, resulting in a possibly still broken proof* π_1.

To unblock π_2*, one of the following unblocking atomic repairs must have been applied to a resolution or factoring step.*

- *Make* $F(\vec{s}^m) = G(\vec{t}^n)$ *to the failed unification problem* $F(\vec{s}^m) \equiv G(\vec{t}^n)$.
- *Make* $x \notin \mathcal{V}(t)$ *to the failed unification problem* $x \equiv t$ *(or* $t \equiv x$*).*

Suppose, wlog, that Unblock had unblocked the unification of L_1 *and* L_2*, so that* σ *now unifies them. Suppose this proof* π_1 *is submitted to Block. It is able to use either of the following atomic repairs to block this successful unification.*

- *Make* $F(\vec{s}^m) \neq F(\vec{t}^m)$ *to the successful unification problem* $F(\vec{s}^m) \equiv F(\vec{t}^m)$.
- *Make* $x \in \mathcal{V}(t)$ *to the successful unification problem* $x \equiv t$ *(or* $t \equiv x$*).*

Similarly to the previous case, each of these blocking atomic repairs is the dual of one of the unblocking atomic repairs. So, one of these two repairs can exactly undo the blocking repair and restore the unification problem to its previous failed state. As before, there are multiple realisations of these blocking steps, but we only need to consider the one that exactly reverses the previous blocking repair and ignore the others.

We consider the two cases in more detail:

Make $F(\vec{s}^m) = G(\vec{t}^n)$: *$\pi_2$ has been unblocked by renaming G to F in L_2, say, and by changing the arities of F or G or both. The blocking repair Make $F(\vec{s}^m) \neq F(\vec{t}^m)$ can rename F back to G and remove any arguments that were added and put back any that were removed.*

Make $x \notin \mathcal{V}(t)$: *π_1 has been unblocked by removing all of these occurrences of x from t. The blocking repair Make $x \in \mathcal{V}(t)$ can replace all these occurrences of x back into t.*

This concludes the proof. QED

7 Implementation

The reformation algorithm has been implemented in SWI-Prolog [SWI, 1993]. Prolog is well suited to the transformation-rule style in which both the unification algorithm and the reformation algorithm have been presented. Prolog also provides the depth-first search needed to return different complete unblocking repairs on backtracking, as described in §8.3. This implementation has been successfully evaluated on a test set of problems, including those described in detail in §9. It also succeeds on all the development examples in §2.

Fig 1 gives the output of our implementation on two examples: one of blocking and one of unblocking. The blocking example is described in §9.2; the unblocking example is described in §9.4. Note that in the blocking example, there are several additional repairs suggested to the one described in §9.2 and in the unblocking example multiple repairs are needed.

```
* Block unification of eq(x,x)=eq(y,c)
R = (['Make y in Vars(c)'], [insert_var(c, y)]) ;
R = (['Make eq \neq eq:  split eq to eq and eq"'],[split(eq)]) ;
R = (['Make eq \neq eq:  add a different argument to eq'],[add_diff_arg(eq)]) ;

* Unblock unification of
    love(x,x)=love2(y,love_of(y))
R = (['Make love = love2: rename love to love2', 'Make y \not in love_of(y)'],
     [merge(love, love2), remove_ith(love_of, 1)]) ;
R = (['Make love = love2: rename love2 to love', 'Make y \not in love_of(y)'],
     [merge(love2, love), remove_ith(love_of, 1)]) ;
```

Figure 1: **Suggestions output from the reformation program**: *Two sets of suggestions are given: one set to block the successful unification of* `eq(x,x)` *and* `eq(y,c)`*; and one to unblock the unsuccessful unification of* `love(x,x)` *and* `love2(y,love_of(y))`*. Prolog uses the convention that functors start with lower case letters and variables with upper case ones — unfortunately, the inverse of the convention that has been used everywhere else in this paper.*

7.1 Search Control

The reformation algorithm generates a large search space of potential repairs, but the situation is different for blocking and unblocking.

Blocking: A proof can be blocked by inverting any successful step in the application of unification during any of the resolutions steps in the proof. As we saw in §8.3, there are also often multiple ways of blocking a successful unification step. For a large proof or one involving complex expressions, there could be a very large number of blocking options.

Unblocking: The search space for unblocking depends on the input. If a faulty proof is available, e.g., the one due to Cauchy described in §2.3, then checking the soundness of each step of this faulty proof may reveal only a handful of faulty steps, each one of which must be unblocked.

Even if these unblocking steps can be realised in different ways, there will only be a few complete repairs.

On the other hand, if only the finitely failed[10] search space of a true but unprovable conjecture is available, then each of the subgoals on each leaf of this search space is a candidate for unblocking. In a large search space there could again be a lot of unblocking options.

We describe some heuristics we have used to reduce or prioritise this search space. They have each been included in one of our implementations of reformation and used in the examples of §9. Unfortunately, some of these heuristics will defeat the proof of theorem (3). We discuss this effect of these heuristics below.

Disallow functions with different arities: It does not make sense to realise Make $F(\vec{s}^m) \neq F(\vec{t}^m)$ by changing the arities of the functions. If this is done as an alternative to splitting the two occurrences of F then we will overload F, i.e., there will be two distinct versions of F with different arities. This is not only bad practice, but is not allowed in some versions of first-order logic. If it is done *as well* as splitting the Fs, then the repair is no longer minimal[11], as splitting the Fs alone is sufficient to block the proof.

However, this heuristic undermines the proof of theorem (3). Without the ability to change arities, this blocking repair can no longer undo any arity changes in the dual Make $F(\vec{s}^m) = G(\vec{t}^n)$ unblocking atomic repair.

Restrict arity change: Nor does it make sense to realise Make $F(\vec{s}^m) = G(\vec{t}^n)$ by allowing unbounded arity changes, since this introduces an infinite branching point into the repair process. We might want to restrict, for instance, to making the new arity be $max(m, n)$ and just adding arguments to the one with fewer arguments. Similarly, we might not want to allow the new arguments to take arbitrary values. In our implementation, for instance, we used new constants, allowing subsequent reformation steps to replace these constants, as needed, to unblock the proof.

These heuristics also undermine the proof of theorem (3). They might prevent this repair from undoing the work of atomic repair. Make $F(\vec{s}^m) \neq F(\vec{t}^m)$.

Restrict variable insertion: Since Make $x \in \mathcal{V}(t)$ can work by inserting only one x in t, it is non-minimal to insert more. Moreover, we can restrict the placement to a conventional position, say, just a new last argument of the function F in which it is inserted.

These heuristics also undermine the proof of theorem (3). Either of them will prevent it from undoing the work of atomic unblocking repair Make $x \notin \mathcal{V}(t)$, which may have had to remove multiple occurrences of x from some functions and some of these occurrences might not be the last argument.

Restrict variable removal: In [Bundy, 1985], we restrict Make $x \notin \mathcal{V}(t)$ to removing x from skolem functions. This is because this repair can be realised by reordering quantifiers, e.g., moving an existential quantifier to be before a universal quantification of x means that x does not become an argument of the skolem function arising from the existential quantifier when it is clausified.

This heuristic will, however, prevent this unblocking atomic repair from undoing some applications of blocking repairs of Make $x \in \mathcal{V}(t)$.

Protect Functors: It does not make sense to repair some basic functors. It wouldn't usually make sense, for instance, to turn = into a unary functor or rename it. The user can add such functors to a protected set. This set is currently initialised with various arithmetic functors, such as + and ×, and =.

[10] If the search space is not finitely failed, then we don't know whether the conjecture is unprovable. We could, however, terminate it after some resource is exhausted, then treat any of its unresolvable leaves as failed steps that are candidates for unblocking.

[11] This is a different sense of minimal from that established by Theorem 3. Exploring different notions of minimality for reformation is ongoing work.

Blocked doomed occurs-check actuations: Since variables in parent clauses are standardised apart before resolution, the repair suggestion Make $x \in \mathcal{V}(t)$ can only succeed if an application of $Var\ Elim$ (see Table 1) unifies x with a variable in the other parent. For this to happen, x must already occur in the parent being repaired. If this is not the case, then Make $x \in \mathcal{V}(t)$ must fail and the repair suggestion can be pruned without loss of completeness.

Minimise false theorems: Blocking repairs are intended to prevent the proof of a false conjecture. As a side effect they might also block other such unwanted proofs. On the other hand, an unblocking repair, as well as allowing the proof of a true conjecture, might accidentally also allow the proof of a false one. We prefer repairs that minimise the number of false theorems.

Maximise true theorems: Similarly, unblocking repairs are intended to allow the proof of a true conjecture. As a side effect they might also allow the proof of additional true conjectures. On the other hand, a blocking repair, as well as preventing the proof of a false conjecture, might accidentally also prevent the proof of a true one. We prefer repairs that maximise the number of true theorems.

To avoid the problems noted above, theorem (3) can be adapted as follows. Note that we need the unrestricted version of reformation only for the undoing parts of the proof. The initial applications of reformation can use any heuristic restrictions we care to impose. So, we can divide theorem (3) into two implications.

$$\begin{aligned} Block'(\pi_1, \pi_2) &\implies Unblock(\pi_2, \pi_1) \\ Unblock'(\pi_1, \pi_2) &\implies Block(\pi_2, \pi_1) \end{aligned}$$

where $Block'$ and $Unblock'$ are the heuristically restricted version of reformation, and $Block$ and $Unblock$ are the unrestricted versions. This adapted theorem is true even in the presence of heuristics. Note that using the unrestricted versions of reformation will not cause a combinatorial explosion as their choices will be guided by the attempt to match the target (broken) proofs π_1.

8 The Application of Repairs to Theories and Conjectures

Reformation repairs must be propagated from the atomic repairs to the initial theory and/or conjecture. To simplify propagation, we will make a lot of additional restrictions. We hope that it will be possible to remove all or some of these restrictions in future work.

In particular, we will restrict inference to the application of selected linear (SL) resolution [Kowalski & Kuehner, 1971] to Horn clauses [Horn, 1951]. This will enable us to apply repairs directly to initial clauses and obviate the need to track the repairs back through the proof to these clauses.

We will also adopt the restriction, introduced in §7.1, that each function has a unique arity. The restriction is not a serious impediment, since for any function with two arities, we can rename one, to give a new theory with an equivalent semantics to the old one.

We start by briefly defining Horn clauses and SL resolution.

8.1 Horn Clauses

We first introduce Kowalski form, in which all the negative literals in a clause are unnegated and conjoined on the left hand side of an implication, and the positive literals are disjoined on the right hand side. It will be much easier to understand SL resolution using this formalism.

Definition 6 (Kowalski Form) *A clause in* Kowalski form *is a formula:*

$$P_1 \wedge \ldots \wedge P_m \implies Q_1 \vee \ldots \vee Q_n$$

where $m, n \geq 0$ and each P_i and Q_j is a proposition. This is logically equivalent to the more standard clause:

$$\neg P_1 \vee \ldots \wedge \neg P_m \vee Q_1 \vee \ldots \vee Q_n$$

Definition 7 (Horn Clauses) *A* Horn clause *is a clause in which $n = 0$ or $n = 1$. It is convenient to distinguish the following 4 cases:*

Rule: $P_1 \wedge \ldots \wedge P_m \implies Q_1$, *where $m > 0$ and $n = 1$. Q_1 is called the* head *and $P_1 \wedge \ldots \wedge P_m$ the* body *of the rule.*

Assertion: $\implies Q_1$, *where $m = 0$ and $n = 1$.*

Goal: $P_1 \wedge \ldots \wedge P_m \implies$, *where $m > 0$ and $n = 0$.*

Empty: $\implies$, *where $m = 0$ and $n = 0$. We will henceforth use $\implies$ instead of $\square$.*

8.2 Selected Literal Resolution on Horn Clauses

In reformation we are concerned with two different kinds of proof:

Refutation of negated conjecture: Resolution proofs are usually by refutation. The conjecture to be proved is negated and put into clausal form. These clauses are conjoined to the axioms, which have also been put into clausal form. If the empty clause can be proved from this conjunction, the conjecture is a theorem.

In reformation we are concerned to unblock failed proofs of true conjectures.

Inconsistent axioms: If the empty clause can be proved from the axiom clauses alone, then the theory they define has been shown to be inconsistent.

In reformation we are concerned to block successful proofs of the empty clause from just the axioms.

Note that both blocking and unblocking are concerned with attempts to prove the empty clause. This is potentially confusing. In the first case above, proving $\implies$ is *desirable* as it shows that a true conjecture has been proved. In the second case above, proving $\implies$ is *undesirable* as it shows that the axioms are inconsistent. However, it will be convenient below to treat these different cases uniformly.

Note that an inconsistent set of Horn clauses must contain a goal clause or the empty clause. If not, then letting all propositions be true creates a model of the clauses, so they can't be inconsistent. In particular, any non-trivial proof of $\implies$ must contain a resolution with a goal clause. SL refutations can be reorganised so that a resolution between a goal clause and an axiom is the first step.

Definition 8 (Selected Literal Resolution on Horn Clauses) *A refutation by* selected literal resolution on Horn clauses *has the following form.*

- *It consists of a sequence of steps.*
- *At each step, one of the literals in a goal clause is selected and then resolved with either an assertion or the head of a rule.*
- *If the resolution is with an assertion then the effect is to delete the selected literal and apply the unifier of the two parent literals to the whole current goal clause.*
- *If the resolution is with a rule then the effect is to replace the selected literal with the body of the rule and then apply the unifier of the two parent literals to the whole current goal clause.*

The choice of assertion or rule can be backtracked over, but not the choice of selected literal.

Figure 2 shows the shape of these proofs. If this reminds you of Prolog, then that is no coincidence: Prolog is based on this type of resolution.

Our restriction to SL resolution on Horn clauses means that one parent of each resolution is always an initial clause. Also, we don't need a factoring rule; binary resolution is sufficient for completeness.

$$\dfrac{\begin{array}{c} P_1 \wedge \ldots \wedge P_m \implies \\ \vdots \\ P'_1 \wedge \ldots \wedge P'_i \wedge \ldots \wedge P'_{m'} \implies \\ \hline (P'_1 \wedge \ldots \wedge P''_1 \wedge \ldots \wedge P''_{m''} \wedge \ldots \wedge P'_{m'})\sigma \implies \\ \vdots \\ \implies \end{array}}{} \quad P''_1 \wedge \ldots \wedge P''_{m''} \implies Q''_1$$

Figure 2: **The Shape of a Selected Literal Resolution on Horn Clauses Refutation.** *σ is the mgu of Q''_1 and selected literal P'_i. If we allow $m'' \geq 0$ then this proof shape also covers the case where the selected literal is an assertion rather than a rule. The proof consists of a backbone of goal clauses ending with the empty clause. At each stage one of the literals of the current goal clause is resolved with an assertion or a rule.*

If we were to allow non-Horn clauses, then we have to allow *ancestor resolution*, which is when a selected literal is resolved with a literal in an earlier goal clause. Such resolutions are possible, because resolving with non-Horn clauses can introduce propositions into the right hand side of the implication. We want to avoid this as it prevents us limiting repairs to initial clauses. This restriction helps with realising the reformation proposed repair as a change to the theory. If we only repair initial clauses this can be directly implemented, whereas if we repair intermediate clauses we have to propagate these repairs back to initial clauses.

8.3 Restrictions on Atomic Repairs

We can now restrict the operation of the atomic repairs to make propagation trivial. Since one parent of each resolution is always an initial clause, we can restrict the red atomic repairs to *initial clauses*, thus minimising the amount of propagation that needs to take place. Wlog, we will assume that the RHS r of unification problems $l \equiv r$ in Table 3 come from this initial clause and the LHS l comes from the current goal clause. To simplify further, we will consider the theory to be a clausal one, so that the repairs to these initial clauses are the end point of propagation.

Unfortunately, the same restriction will not work for the two blue atomic repairs. Note that the **Before** column of transformation rules VC_s and VC_f contains both $x \equiv t$ and $t \equiv x$, so either x or t could come from the initial clause, but the repair must be to t, i.e., either adding x as an extra argument to some function h that occurs in t or removing an argument of h that contains x. Wlog, if we have to add x to h, we can make this a new last argument of h. Fortunately, our restriction on allowing a function to have more than one arity means that all occurrences of h in the proof or initial clauses[12] must have the same arguments either added or removed.

We now consider the required restrictions on each of the atomic repairs in turn.

Make $F(\vec{s}^m) \neq F(\vec{t}^m)$: As suggested in §7.1, we restrict this repair to the renaming of *one* occurrence of F. Since the RHS $F(\vec{t}^m)$ term occurs in an initial parent clause, we rewrite this term to $F'(\vec{t}^m)$, where F' is a new m-ary function name. Note that this initial clause may be used elsewhere in the proof, so this change may block other proof steps too.

Make $F(\vec{s}^m) = G(\vec{t}^n)$: Again, we restrict the repairs to the term $G(\vec{t}^n)$ occurring in an initial clause. We rename G to F, but we also need to change its arity to m by either adding or removing arguments.

[12] In case one of the initial clauses is not used in the proof.

If $n > m$ we need to remove $n - m$ arguments, but which ones? We perform a heuristic alignment of the s_i with the t_j and use this to decide which t_j to keep. Optionally, we can reordering the t_j to improve the alignment.

If $m > n$ we need to add $m - n$ arguments. Again, we can perform a heuristic alignment to decide which of the s_i do not correspond to any t_j and then add arguments to G to fill those gaps. We might could short circuit reformation by using the missing s_i as argument fillers, but that might not suit other applications of this initial clause, so we adopt the safer option to gen-sym up some new constants and then let subsequent applications of reformation decide what to reform these constants to.

Make $x \in \mathcal{V}(t)$: Choose, non-deterministically, a term $H(r_1, \ldots, r_k)$ occurring in t. For all occurrences of H in the proof or initial clauses, add x as a new last argument. Note that the *blocked doomed occurs-check actuations* heuristic of §7.1 may veto this repair if it can't be effective.

Make $x \notin \mathcal{V}(t)$: Choose, non-deterministically, a term $H(r_1, \ldots, r_k)$ occurring in t for which some r_i contains x. Remove the i^{th} argument from all occurrences of H in the proof or initial clauses. Repeat this step until t no longer contains any occurrence of x.

9 Worked Examples

To further confirm the hypothesis in §1, reformation has been successfully evaluated on an extensive test set. Space limitations prohibit discussion of the whole test set, so we have selected just one example for each of the atomic repairs in §8.3. We have divided them into blocking and unblocking examples. On each of these examples, reformation gives the results discussed below.

To avoid ambiguity, when describing unwanted unification failures or successes, we will refer to steps in the original algorithms in Table 1. This ambiguity arises because we have used the same names for steps in both the non-standard unification algorithm (Table 2) and the reformation algorithm (Table 3).

We now give two examples of blocking an application of unification to defeat the proof of a false theorem.

9.1 Repairing Family Relations using Make $F(\vec{s}^m) \neq F(\vec{t}^m)$

Motherhood has become a complex relationship. Instead of a simple function from each child to a unique birth mother, we now have stepmothers, foster mothers, surrogate mothers, etc. This ambiguity is likely to lead to faulty theories, such as the one illustrated by the following example.

Consider the following inconsistent set of axioms in clausal form:

$$\begin{array}{l|l} \Longrightarrow\ Mum(Diana, William) & \Longrightarrow\ Mum(Camilla, William) \\ Diana = Camilla\ \Longrightarrow & Mum(m_1, c) \wedge Mum(m_2, c)\ \Longrightarrow\ m_1 = m_2 \end{array}$$

Suppose reformation is used to block the following derivation of the empty clause $\Longrightarrow$ at the highlighted step.

$$\dfrac{\dfrac{\dfrac{Diana = Camilla\ \Longrightarrow}{Mum(Diana, c) \wedge Mum(Camilla, c)\ \Longrightarrow}}{\dfrac{Mum(Camilla, William)\ \Longrightarrow}{\Longrightarrow}}}{\begin{array}{l} Mum(m_1, c) \wedge Mum(m_2, c)\ \Longrightarrow\ m_1 = m_2 \\ \Longrightarrow\ Mum(Diana, William) \\ \Longrightarrow\ Mum(Camilla, William) \end{array}} \quad (3)$$

Step CC_s of the reformation algorithm is triggered and suggests the following two repairs to the axiom $\Longrightarrow Mum(Camilla, William)$: either rename Mum or add an additional argument. Another suggestion is made during a recursive application of CC_s. All three suggestions have intuitive interpretations.

- Mum might be renamed, say, to Mum_1. This would make sense, for instance, if the new predicate were subsequently interpreted, by the user or some third party program, as $StepMum$.
- Mum might be given an additional argument. This new argument might also subsequently be interpreted, for instance, as indicating the kind of motherhood, e.g., step-motherhood or birth motherhood.
- During a recursive call to CC_s, one occurrence of $William$ might be renamed to $William_1$ or be given an additional argument. These repairs would make sense if two different individuals were both named William and needed to be distinguished.

Reformation will not, of course, suggest such meaningful names for these new functors or arguments. They could only be suggested by a user or some domain-specific program, but the potential to do so is suggested by reformation. We have used such meaningful names below to aid readability.

The first suggestion will then revise the previously inconsistent theory to the consistent theory:

$$\begin{array}{l|l} \implies Mum(Diana, William) & \implies StepMum(Camilla, William) \\ Diana = Camilla \implies & Mum(m_1, c) \wedge Mum(m_2, c) \implies m_1 = m_2 \end{array}$$

The other two repairs are left as an exercise for the reader.

Unfortunately, reformation will also suggest some less intuitive repairs. For instance, during a recursive call, the variable m might be added as an argument to $Camilla$, to induce an occurs check failure. This repair does not actually block the unification, because the standardising apart of the variables in parent clauses will mean that there will be no attempt to unify m with $Camilla(m)$. Fortunately, this unsuccessful suggestion is pruned by the 'Blocked doomed occurs check actuations' heuristic discussed in §7.1.

Also, of course, the unwanted proof (3) could be blocked at any of the other proof steps. For instance, blocking it at the second step would suggest similar repairs as above, but to the axiom $\implies Mum(Diana, William)$ rather than $\implies Mum(Camilla, William)$. Blocking at the first step would suggest changing the name or arity of $=$, but this repair is disallowed because $=$ is a protected functor (see §7.1).

9.2 Repairing a Faulty Equality Theory using Make $x \in \mathcal{V}(t)$

Suppose a theory of equality contains the faulty axiom:

$$\exists z, \forall y.\ y \neq z \tag{4}$$

which is clearly false in any non-trivial, standard model of equality. After skolemisation, this yields the goal clause $y = C_1 \implies$, where C_1 is a new skolem constant arising from the existential variable z. Resolution of this clause with the reflexivity axiom then gives the empty clause, showing that the faulty equality theory is inconsistent:

$$\frac{y = C_1 \implies}{\implies} \implies x = x$$

where the mgu is $\{y/C_1, x/C_1\}$.

Reformation suggests blocking this resolution step by adding y as an additional argument of c. The proof above is thereby transformed into a broken proof. x is first substituted by y in an application of unification step $Var\ Elim$. The unification sub-problem of $y \equiv C_1(y)$ is then generated, which fails the occurs check, blocking the unification and preventing the resolution, as desired.

$$\frac{y = C_1(y) \implies}{Blocked} \implies x = x$$

Note that the 'Blocked doomed occurs check actuations' heuristic is not triggered in this case, as a prior application of *Var Elim does* unify x and y.

The repair effectively reverses the order of the quantifiers, changing the faulty axiom (4) into

$$\forall y, \exists z.\ y \neq z$$

which is true in any non-trivial theory. For instance, if the domain were integers, then one witness for z is $y + 1$.

We now give two examples of unblocking an unsuccessful application of unification to enable the proof of a true conjecture.

9.3 Repairing More Family Relations using Make $F(\vec{s}^m) = G(\vec{t}^n)$

The multiple kinds of parenthood can be represented either by having different predicates for each kind, e.g., $StepMum$, $FosterMum$, etc. or by adding an additional argument to relevant predicates to indicate what kind of parenthood is intended, e.g., adding a $Step$ or $Foster$ argument to Mum or $Parent$. If different solutions are adopted by different theory co-designers, then it may not be possible to prove some true conjectures in the resulting theory.

Consider the following incomplete set of axioms in clausal form:

$$\Longrightarrow StepMum(Camilla, William) \;\Big|\; StepMum(p, c) \Longrightarrow Parent2(p, c, Step)$$

together with the goal clause $Parent(Camilla, William) \Longrightarrow$.

Consider the following failed refutation from that goal clause:

$$\frac{Parent(Camilla, William) \Longrightarrow}{Blocked} \quad StepMum(p, c) \Longrightarrow Parent2(p, c, Step)$$

When unification is applied to $Parent(Camilla, William) \equiv Parent2(p, c, Step)$ it fails at case *Clash* because the first occurrence of $Parent$ has 2 arguments and the second has 3, and because the functor names don't agree.

Reformation step CC_f suggests making $Parent(Camilla, William) = Parent2(p, c, Step)$, which requires us to merge the functor names and equate the arities. To merge the functor names, $Parent2$ can be renamed to $Parent$, or vice versa. Equating the arities can be implemented in many ways, e.g., adding a new constant, say C_2, as an extra argument to the binary occurrence of $Parent$ or removing an argument from the ternary occurrence.

If the path of repairing goal clause $Parent(Camilla, William) \Longrightarrow$ to $Parent2(Camilla, William, C_2) \Longrightarrow$ is chosen, unification subsequently fails at case *Clash* on $C_2 \equiv Step$. Reformation step CC_f then suggests making $C_2 = Step$, which can be implemented by renaming C_2 to $Step$ (or vice versa). This gives the successful refutation:

$$\frac{\dfrac{\Longrightarrow Parent2(Camilla, William, Step)}{StepMum(Camilla, William) \Longrightarrow}}{\Longrightarrow} \quad StepMum(p, c) \Longrightarrow Parent2(p, c, Step)$$
$$\Longrightarrow StepMum(Camilla, William)$$

as desired.

The axioms are unchanged by this repair but the goal clause has been changed from $Parent(Camilla, William) \Longrightarrow$ to $Parent2(Camilla, William, Step) \Longrightarrow$.

9.4 Repairing Self Love using Make $x \notin \mathcal{V}(t)$

The sentence "Every man loves a women" is well known to be ambiguous: (a) is there just the one women that every man loves, or (b) for each man, is there some women for him to love? For simplicity we will ignore the male/female distinction. Under interpretation (a) it should then be

possible to prove that someone loves themselves, i.e., $\exists x.\ Loves(x, x)$, but this cannot be proved from interpretation (b). Interpretation (b) can be represented as:

$$\forall x.\exists y.\ Loves(x, y) \tag{5}$$

which in clausal form is $Loves(y, Love\_Of(y))$, where $Love\_Of$ is a new skolem function, which I have given a meaningful name to assist interpretation.

If we try to prove that a self-lover exists using interpretation (b), the attempted proof will fail, e.g., at the following resolution step:

$$\frac{Loves(x, x) \implies}{Blocked} \implies Loves(y, Love\_Of(y))$$

The cause of the failure is due to $Occurs$, as there will be an attempt to unify x with $Love\_Of(x)$. Reformation step VC_f suggests making $x \notin \mathcal{V}(Love\_Of(x))$, which can be done by removing the argument of $Love\_Of$, which can be realised by repairing axiom (5) to:

$$\exists y.\forall x.\ Loves(x, y) \tag{6}$$

which is now interpretation (a). The previously blocked inference step will now succeed.

$$\frac{Loves(x, x) \implies}{\implies} \implies Loves(y, C_3)$$

where C_3 is a new skolem constant representing this remarkable person that everyone loves.

10 Related Work

Several different fields of AI have investigated the problem of changing representations.

Belief revision deals with the problem of adding a new belief to a theory that might result in it becoming inconsistent [Gärdenfors & Rott, 1995]. In this case, mechanisms have been designed to reestablish its consistency. These involve the identification of facts or (more rarely) rules to be deleted. Our work is complementary to this in suggesting changes to the language instead of the deletion of axioms.

Abduction deals with the problem of identifying explanatory hypotheses for observations [Cox & Pietrzykowski, 1986]. New facts or rules are identified as hypotheses which, when added as new axioms to a theory, enable the observations to be deduced. Again, our work is complementary to this in suggesting changes to the language instead of the deletion of axioms.

Ontology evolution deals with the problem of managing an ontology[13]. Most of the work in this area consists of tools to assist manual construction and maintenance of ontologies. Reformation differs in providing a tool for *automatic* identification of changes to the *signature* of an ontology. There is some work on *automated* ontology evolution that also suggests language changes, e.g., [McNeill & Bundy, 2007, Lehmann *et al*, 2012], which apply to the domains of plan formation and physics theories, respectively. Our work builds on this work, but differs in describing a *domain-independent* mechanism that suggests language changes both to block and unblock unifications during *any* deduction that uses unification.

11 Further Work

The development of reformation is a huge work programme, which we have only just begun to explore. Some of our plans for further exploration are outlined in this section.

[13] `http://www.w3.org/TR/webont-req/\#goal-evolution`

11.1 Additional kinds of theory repair

The possibilities for theory repair are open ended. §8.3 only covers some of them. Other possibilities include adding or deleting axioms. Blocking can also be realised by deleting an axiom used in the proof and unblocking by adding an unproven sub-goal as an axiom. In this way reformation, belief revision and abduction can be combined.

11.2 Heuristics for search control

§7.1 describes some of the heuristics we have implemented to control search through the huge space of potential repairs. This section describes some additional possible heuristics.

Functor Prioritisation: §7.1 describes how some functors are protected. Rather than provide absolute protection, we might instead impose a prioritisation partial order on functors, so that functors with a lower priority are repaired in favour of those with a higher priority.

Fact Preference: Ground, atomic axioms, or *facts*, are like data and rules are like universal laws or even definitions. We might, therefore, consider facts more likely to be faulty, so restrict or prioritise repairs to facts.

Repair only False Clauses: If we have access to an oracle that can identify false clauses, then we could restrict reformation just to the repair of such clauses.

Avoid Redundant Clauses: Some false clauses can be derived in multiple ways. Thus blocking one proof of such a clause would not be sufficient; additional repairs would be required to defeat all its other proofs. Thus, if there are other blocking options available, then repairing redundant clauses might be a low priority. Alternatively, one might first work to create an irredundant axiom set before making any repairs.

11.3 Extending reformation to other inference systems

Lifting the restriction to SL resolution on horn clauses: In §8, we restricted reformation to SL resolution on horn clauses in order to guarantee that one parent of each resolution was an initial clause. In this way we avoided the problem of propagating changes through a proof to axioms. By developing a change propagation algorithm, we could lift this restriction.

Adapting reformation to other logics: We have already adapted reformation to many-sorted logics and description logics. It can potentially be adapted to any logic that uses a unification algorithm during inference.

Other unification algorithms: Some unification algorithm build in axioms, such as associative-commutative unification, so versions of reformation might be based on such unification algorithms. Higher-order unification also builds in such axioms, so reformation might be extensible to type theory and other higher-order logics.

12 Conclusion

We have proposed a new algorithm, reformation, for systematically suggesting changes of language to repair a faulty logical theory. The faults are revealed either by a failure to prove a true conjecture or by a success in proving a false conjecture. The language changes are to split one functor into two or merge two into one, or to change the arity of a functor. We have proved unification to be both sound and self-inverse. These results support our hypothesis that:

> *Reformation systematically generates reversible theory repairs that invert the outcome of unification.*

Reformation works by blocking or unblocking an attempted unification. It uses a non-standard unification algorithm in which failing steps are paired with succeeding steps. The changes are designed to flip from one member of a pair to the other. We have shown that this non-standard unification is functionally equivalent to the standard one. The reformation algorithm has been implemented and successfully evaluated in SWI-Prolog. Various heuristics have been suggested to reduce the size of the, otherwise large, search space. Some of these heuristics have been implemented.

Many of the repairs suggested are intuitively plausible and result in an acceptable refinement of the theory's language. These language repairs complement the more usual axiom deletions and additions that are traditionally used in belief revision and abduction. They provide a way in which an axiom is not totally lost, but has its meaning evolved. Both these mechanisms are needed. In further work we will explore how to combine our language-changing repairs with axiom addition/deletion ones.

The mechanism has been proposed in the context of resolution proofs using first-order clauses. It can be (and has been) adapted to any inference mechanism that uses a unification algorithm.

The reformation algorithm is generic and could find a number of applications. Amongst these are:

- As part of a theory development tool, to suggest revisions to a developing or evolving theory.
- As part of a debugging tool for declarative programs, to identify typos that are preventing programs from succeeding or causing programs to succeed with inappropriate results.

References

[Ascoli, 1883] Ascoli, G. (1883). Le curve limiti di una varietà data di curve. *Atti della R. Accad. Dei Lincei Memorie della Cl. Sci. Fis. Mat. Nat.*, 18(3):521–586.

[Baader & Snyder, 2001] Baader, F. and Snyder, W. (2001). Unification theory. In Robinson, J. A. and Voronkov, A., (eds.), *Handbook of Automated Reasoning, Volume 1*, volume I, chapter 8, pages 447–553. Elsevier.

[Bundy, 1985] Bundy, A. (1985). Poof analysis: A technique for concept formation. In Ross, P., (ed.), *Proceedings of AISB-85*, pages 78–86. Society for the Study of Artificial Intelligence and Simulation of Behaviour.

[Cox & Pietrzykowski, 1986] Cox, P. T. and Pietrzykowski, T. (1986). Causes for events: Their computation and applications. In Siekmann, J., (ed.), *Lecture Notes in Computer Science: Proceedings of the 8th International Conference on Automated Deduction*, pages 608–621. Springer-Verlag.

[Etzioni *et al*, 2011] Etzioni, O., Fader, A., Christensen, J., Soderland, S. and Mausam, M. (2011). Open information extraction: The second generation. In *Proceedings of IJCAI*, pages 3–10.

[Gärdenfors & Rott, 1995] Gärdenfors, P. and Rott, H. (1995). Belief revision. In Gabbay, D., Hogger, C. J. and Robinson, J.A., (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, pages 35–132. Oxford Science Publications, Oxford, New York.

[Gkaniatsou *et al*, 2012] Gkaniatsou, A., Bundy, A. and McNeill, F. (November 2012). Towards the automatic detection and correction of errors in automatically constructed ontologies. In *8th International Conference on Signal Image Technology and Internet Based Systems*, pages 860–867, Sorrento, Italy. IEEE.

[Horn, 1951] Horn, Alfred. (1951). On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21.

[Kowalski & Kuehner, 1971] Kowalski, R. A. and Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence*, 2:227–60.

[Lakatos, 1976] Lakatos, I. (1976). *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press.

[Lehmann *et al*, 2012] Lehmann, J., Chan, M. and Bundy, A. (2012). A higher-order approach to ontology evolution in Physics. *Journal on Data Semantics*, pages 1–25.

[McNeill & Bundy, 2007] McNeill, F. and Bundy, A. (2007). Dynamic, automatic, first-order ontology repair by diagnosis of failed plan execution. *International Journal On Semantic Web and Information Systems*, 3(3):1–35. Special issue on ontology matching.

[Mitrovic, 2013] Mitrovic, B., (2013). Repairing inconsistent ontologies using adapted reformation algorithm for sorted logics. UG4 Final Year Project, University of Edinburgh.

[Robinson, 1965] Robinson, J. A. (1965). A machine oriented logic based on the resolution principle. *J Assoc. Comput. Mach.*, 12:23–41.

[Schulz, 2002] Schulz, S. (2002). E — a brainiac theorem prover. *AI Communications*, 15(2-3):111–126.

[Sutcliffe *et al*, 1994] Sutcliffe, G., Suttner, C. and Yemenis, T. (1994). The TPTP problem library. In Bundy, Alan, (ed.), *12th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 252–266, Nancy, France. Springer-Verlag.

[SWI, 1993] Dept. of Social Science Informatics (SWI), University of Amsterdam. (1993). *SWI-Prolog 1.6 Reference Manual*.

[Tsialos, 2015] Tsialos, A., (2015). Repairing inconsistent description logic ontologies using reformation. UG4 Final Year Project, University of Edinburgh.